Slide 1

Administrivia

- (None?)

Slide 2

Minute Essay From Last Lecture

- (Review MIPS problem — but most people got it right.)
- One person, however, thought the code after `label:` was skipped since it's not referenced. Not so! not like a function in HLL, which would need to be called; instead execution just proceeds sequentially.
- More than one person pointed out the typo in the last problem (reverse compiling). I wish someone had said something in e-mail!
- Not much really stood about comments on Homework 2. A few people mentioned that it was challenging (good?), others that they did understand the material better after doing it (good!).

Slide 3

MIPS Assembly Programs — Examples

- (Look briefly at "read and echo integer" example.)
- What would code for a recursive factorial function look like?
- One thing we need is an instruction to multiply. That turns out to be a little complicated, so for now punt and use `mul` pseudoinstruction, which takes three register numbers.
- (Write code to define and test.)

Slide 4

From Source Code to Execution — Recap/Review

- Four main phases, conceptually at least — compile, assemble, link, load.
- Real systems (or simulators) may combine steps, in appearance or even in reality — e.g., a compiler might go directly from high-level source to object code, in appearance or in fact, and the SPIM simulator assembles "on the fly".

Slide 5

Compiling — Review

- As we've seen, translating from HLL source to assembly is not trivial.
- One reason compilers are so big and complicated, however, is that more and more they try to "optimize" (generate code that's more efficient than a naive translation).
- As an example: Textbook goes into some detail about compiling C code to loop through an array, showing a version that uses indices and one that uses pointers. A "good" compiler will likely generate the same code for both.

  Can test this with `gcc`: `-S` generates (x86) assembler, so we could write it both ways, compile to assembler, and compare. Same *if* compiled with `-O3`.

Slide 6

Assembling — Review

- Job of the assembler is to produce "object code". Details might vary among platforms, but several basic parts: header information including sizes of code and data segments, actual machine instructions, table of symbols defined, and table of unresolved references.
- ("Code and data segments"? the former is machine code (`.text` in MIPS assembly), the latter fixed-at-compile-time data (`.data` in MIPS assembly. If you wonder "as opposed to what?" about that fixed-at-compile time — other choices for where to keep data are on a stack and in some other area ("on the heap").

Slide 7

Assemblers — How They (Could?) Work

- (I admit I have not looked at actual code for an assembler, but the job seems straightforward.)
- First start by establishing starting addresses for code and data segments.
- As each instruction or data declaration is encountered, add to appropriate segment and increment "next" address. Also build "symbol table" of labels versus addresses and list of references to labels and make note of any declared as "global".
- Resolve references using symbol table; make list of any that can't be resolved.
- Output sizes, code and data segments, and lists of symbols and unresolved references. (Also, I think, need to output list of instructions using absolute addresses, since these have to be changed at link time.)

Slide 8

Linking — Review

- Job of linker is combine one or more object files into "executable file". Details vary among platforms, but must include anything the operating system needs to load the program into memory and start it up — sizes of code and data segments, location of starting address, anything that needs to be resolved/fixed at runtime.
- So at a minimum, linker must:
  - Merge tables of "global" symbols into combined symbol table.
  - Use it to resolve unresolved references.
  - Merge code segments, data segments.
  - Modify any absolute addresses.
  - Output executable file.

  (All really kind of common sense given goal, no?)

Slide 9

Loaders — Recap/Review

- Job of loader is to load executable file into memory and start it up.
- Exactly what's involved depends on platform, but at a minimum has to read the program into memory and transfer control to starting address.
- Other things might include modifying absolute addresses to reflect actual location in memory (necessary, e.g., if many programs in the same "address space"), resolving references to dynamically-linked library code.
- As an example of this and also of combining steps, consider what must happen with you load a (source) program into SPIM: It assembles (on the fly), loads the result object code into memory, linking it (on the fly) with the tiny bit of startup code that does a `jal` to `main`, and starts the startup code.

Slide 10

Linking — Example

- (Work through example starting on p. 127 — next time.)

Slide 11

Minute Essay

- Any questions?